Chapter 5

The Algebra of Programs

Our study of the theory of programming started in Chapter 2 with a very simple definition within the predicate calculus of the notations of our programming language. Of course, the notations of predicate calculus are much more powerful than those of any implementable programming language – they even include the contradictory predicate **false**. In Chapter 3, a series of healthiness conditions were introduced to characterise more precisely those predicates which are expressible, or at least implementable, in the programming language. Each restriction on the class of predicates was motivated and rewarded by the proof of validity of an additional algebraic law. In this way it was possible to resolve the paradoxes of the first naive attempt at programming language definition.

In this chapter, we adopt a different strategy, which yields an even more precise characterisation of the class of programs and an even more complete collection of properties, including the important concepts of continuity. We start by *defining* a program as a predicate actually expressed (or at least expressible) in the limited notations of the programming language, as summarised in Table 5.0.1. Mathematicians often use similar notational definitions to single out simple classes of functions. For example, the class of polynomials can be defined as functions expressible from just constants and variables, combined by arithmetic operators of addition, subtraction and multiplication.

The next step is to list a collection of laws expressing familiar properties of the chosen operators, and their mutual interactions. These laws can be sufficiently powerful to reduce every expression in the restricted notation to an even more restricted notation, called a *normal form*. For example, it is well known that all polynomials can be reduced to a Horner normal form, whose syntax is defined by a simple recursion

$$< HNF > \ ::= \ < constant > \ | (< HNF >) * < variable > + < constant >$$

Additional properties of polynomials can now be simply deduced by showing them to be valid just for normal forms. Similarly we shall define and use a normal form for programs to prove the validity of the zero laws for our programming language. Once the normal form is established there is no need to look for healthiness conditions that place semantic restrictions on the range of predicates under consideration. These too emerge from analysis of the normal forms. Above all, there is no need to introduce extra variables like ok and ok', which can be regarded as artificial coding tricks, and which evoke the philosophical objection that (because of non-termination) they can never be observed to be false.

<program> :: = **true**
| <variable list> := <expression list>
| <program> $\lhd$ <Boolean expression> $\rhd$ <program>
| <program> ; <program>
| <program> $\sqcap$ <program>
| <recursive identifier>
| μ <recursive identifier> $\bullet$ <program>

Table 5.0.1 Syntax

A new objective in this chapter is to bring some order into the large collection of laws about programs that have been proved in previous chapters. The method is to identify a minimum collection of laws from which all the other laws can be deduced by algebraic reasoning. Reduction in the number and complexity of independently postulated laws is one of the primary goals of theoretical investigation in any branch of the natural sciences. Success not only appeals to a sense of intellectual elegance; it also leads to new insight, which unifies understanding, and broadens it to new areas of application. The same goal is sought in the more abstract study of algebra, and for the same reasons. Abstract algebra is clearly the most reusable branch of mathematics, but each time that it is reused, it is necessary to prove that the laws are in fact true in the new application domain. This is much easier if a minimum collection of laws has been identified in advance. Then only these few laws have to be checked, because the rest have already been proved to follow from them. That is why emphasis will constantly return to the question of the completeness of the laws presented.

This chapter starts with a series of increasingly complex and general normal forms, dealing successively with assignment, non-determinism, non-termination and recursion, of which iteration (Section 5.5) is an important special case. In

Sections 5.6 and 5.7 an argument is given for the *computability* of the normal form, and for the completeness of the algebraic presentation of the language as a whole. We derive a denotational semantics from the algebraic in Section 5.8.

5.1 Assignment normal form

The first in our series of normal forms is the total assignment, in which all the variables of the program appear on the left hand side in some standard order

$$x, y, \ldots, z \;:=\; e, f, \ldots, g$$

A non-total assignment can be transformed to a total assignment by addition of identity assignments $(z, \ldots \;:=\; z, \ldots)$

L1 $(x, y, \ldots \;:=\; e, f, \ldots) \;=\; (x, y, \ldots, z \;:=\; e, f, \ldots, z)$

The list of variables may be sorted into any desired order, provided that the right hand side is subjected to the same permutation

L2 $(x, \ldots, y, z, \ldots \;:=\; e, \ldots, f, g, \ldots) \;=\; (x, \ldots, z, y, \ldots \;:=\; e, \ldots, g, f, \ldots)$

As mentioned in Chapter 2, we abbreviate the entire list of variables $(x, y, \ldots, z)$ by the simple vector variable v, and the entire list of expressions by the vector expressions $g(v)$ or $h(v)$; these will usually be abbreviated to g or h. Thus the assignment normal form will be written

$$v \;:=\; g \quad \text{or} \quad v \;:=\; h(v)$$

The law that eliminates sequential composition between normal forms is

L3 $(v \;:=\; g;\; v \;:=\; h(v)) \;=\; (v \;:=\; h(g))$

The expression $h(g)$ is easily calculated by substituting the expressions in the list g for the corresponding variables in the list v. For example,

$$(x, y \;:=\; x+1, y-1 \;;\; x, y \;:=\; y, x) \;=\; (x, y \;:=\; y-1, x+1)$$

To deal with the conditional combinator, we now need to assume that our programming language allows conditional expressions on the right hand side of an assignment. Such an expression is defined mathematically

$$\begin{aligned} e \lhd c \rhd f \;&=_{df}\; e && \text{if } c \\ &=_{df}\; f && \text{if } \neg c \end{aligned}$$

The definition can be extended to lists, for example

$$(e1, e2) \lhd c \rhd (f1, f2) \quad =_{df} \quad ((e1 \lhd c \rhd f1), (e2 \lhd c \rhd f2))$$

Now the elimination law for conditionals is

L4 $((v := g) \lhd c \rhd (v := h)) \;=\; (v := (g \lhd c \rhd h))$

Finally, we need a law that determines when two differently written normal forms are equal. For this, the right hand sides of the two assignments must be equal for all values of the variables that they contain

L5 $(v := g) = (v := h)$ **iff** $[g = h]$

It is this last law that justifies a claim that the entire collection **L1** to **L5** is *complete*, in the sense that they are sufficient to prove any valid equations between two programs that have been expressed in the programming notations of assignment, sequential composition and conditional. First, the laws **L1** to **L4** are used to reduce both sides of the equations to assignment normal forms, and then **L5** replaces the two normal forms by an equation that contains no programming notations whatsoever. The original equation between programs is valid just if the transformed program-free equation is valid. A proof constructed in this way is called an *algebraic proof*, and any equation (or inequation) that has such a proof is said to be algebraically provable. Of course, if g and h in **L5** are expressions of an incomplete logic, the algebra of programs will be equally incomplete. This means that a kind of relative completeness has to be accepted as the best that can be achieved in a calculus of programming.

Example 5.1.1

$(x := y - x \lhd x \leq y \rhd y := x)\,;\, (x := x + 3) \;=$

$(y := y \lhd x \leq y \rhd x)\,;\, (x := (y - x + 3) \lhd x \leq y \rhd (x + 3))$

Proof

$$\begin{aligned}
& LHS && \{\textbf{L1}, \textbf{L2}\} \\
= \;& (x, y := y - x, y) \lhd x \leq y \rhd (x, y := x, x)\,; \\
& x, y := x + 3, y && \{\textbf{L4}\} \\
= \;& x, y := (y - x \lhd x \leq y \rhd x), (y \lhd x \leq y \rhd x)\,; \\
& x, y := x + 3, y && \{\textbf{L3}\} \\
= \;& x, y := (y - x \lhd x \leq y \rhd x) + 3, (y \lhd x \leq y \rhd x)
\end{aligned}$$

We now reduce the other side to normal form

$$
\begin{array}{lll}
 & RHS & \{\mathbf{L1}, \mathbf{L2}\} \\
= & x, y := x, (y \lhd x \leq y \rhd x)\,; & \\
 & x, y := (y - x + 3 \lhd x \leq y \rhd x + 3), y & \{\mathbf{L3}\} \\
= & x, y := (y - x + 3 \lhd x \leq y \rhd x + 3), y \lhd x \leq y \rhd x &
\end{array}
$$

By **L5** the original equation is equivalent to a statement of the equality of the assigned expressions in these two normal forms, that is

$$
\begin{array}{l}
(y - x \lhd x \leq y \rhd x) + 3, (y \lhd x \leq y \rhd x) = \\
(y - x + 3 \lhd x \leq y \rhd x + 3), (y \lhd x \leq y \rhd x)
\end{array}
\qquad \square
$$

It is possible to use this kind of algebraic reasoning to prove more useful and more general laws, for example the idempotence law for the conditional

$$P \lhd b \rhd P \;=\; P$$

First, P can be reduced to the assignment normal form, so what needs to be proved is only the special case

$$(v := e) \lhd b \rhd (v := e) \;=\; (v := e)$$

Then law **L4** shows this is equivalent to

$$v := (e \lhd b \rhd e) \;=\; (v := e)$$

Law **L5** requires us to prove

$$[e \lhd b \rhd e \;=\; e]$$

which contains no programming notations, and follows directly from the definition of the conditional expression. Of course, this argument is *schematic*: it is not itself a formal proof, but it shows how a proof is constructed in each particular case.

Reduction of particular programs to normal form is an activity requiring no insight, and so it is more suited for delegation to a machine. For human reasoning and understanding, it is usually better to bypass the normal form, and use a much larger collection of algebraic laws, which have been proved from the minimal set, usually by schematic proof. Minimisation of the number of laws that have to be proved, remembered or implemented on a computer is a good general goal, but it does not obviate the need for a much larger collection of laws in the application of algebra to practical reasoning.

5.2 Non-determinism

Disjunction between two semantically distinct assignments cannot be reduced to a single assignment, which is necessarily deterministic. We therefore move to a more complicated normal form, in which the disjunction operator connects a finite non-empty *set* of total assignments

$$(v := f) \sqcap (v := g) \sqcap \ldots \sqcap (v := h)$$

Let A and B be such sets; we will write the normal form as $\bigsqcap A$ and $\bigsqcap B$. All the previous normal forms can be trivially expressed in the new form as a disjunction over the unit set

$$v := g \;=\; \bigsqcap\{v := g\}$$

The easiest operator to eliminate is disjunction itself; it just forms the union of the two sets

$$(\bigsqcap A) \sqcap (\bigsqcap B) \;=\; \bigsqcap(A \cup B)$$

The other operators are eliminated by distribution laws

L1 $(\bigsqcap A) \lhd b \rhd (\bigsqcap B) \;=\; \bigsqcap\{(P \lhd b \rhd Q) \mid P \in A \wedge Q \in B\}$

L2 $(\bigsqcap A);(\bigsqcap B) \;=\; \bigsqcap\{(P;Q) \mid P \in A \wedge Q \in B\}$

The right hand sides of these equations are disjunctions of terms formed by applying the relevant operator to total assignments P and Q, which have been selected in all possible ways from A and B. Each of these terms can therefore be reduced to a total assignment, using the laws of Section 5.1. Thus the occurrences of ; and $\lhd b \rhd$ in the right hand sides of the laws given above are also eliminable.

The laws which permit comparison of disjunctions are

L3 $(\bigsqcap A) \sqsupseteq R$ **iff** $\forall P : P \in A \bullet (P \sqsupseteq R)$

L4 $(v := f) \sqsupseteq (v := g \sqcap \ldots \sqcap v := h)$ **iff** $[f \in \{g, \ldots, h\}]$

The first law is a tautology; it enables a disjunction in the antecedent to be split into its component assignments, which are then decided individually by the second law.

Exercise 5.2.1

Extend to this more general normal form the schematic proof in the previous section of

$$P \lhd b \rhd P \;=\; P$$ □

5.3 Non-termination

The program constant **true** is not an assignment, and cannot in general be expressed as a finite disjunction of assignments. Its introduction into the language requires a new normal form

$$\mathbf{true} \lhd b \rhd P$$

where P is in the previous normal form. It is more convenient to write this as a disjunction

$$b \vee P$$

Any unconditional normal form P can be expressed as

$$\mathbf{false} \vee P$$

and the constant **true** as

$$\mathbf{true} \vee I\!I$$

where $I\!I$ is $v := v$. The other operators between the new normal forms can be eliminated by the following laws

L1 $(b \vee P) \sqcap (c \vee Q) = (b \vee c) \vee (P \sqcap Q)$

L2 $(b \vee P) \lhd d \rhd (c \vee Q) = (b \lhd d \rhd c) \vee (P \lhd d \rhd Q)$

L3 $(b \vee P); (c \vee Q) = (b \vee (P; c)) \vee (P; Q)$

L4 $(\bigsqcap A); b = \bigvee \{(P; b) \mid P \in A\}$

L5 $(v := e); b(v) = b(e)$

As before, the occurrences of each operator on the right hand side can be further reduced by the laws of the previous sections. Laws **L4** and **L5** ensure that the terms $P; c$ and $P; b$ can be reduced to simple conditions.

The laws for testing implication between these new normal forms are

L6 $(b \vee P) \sqsupseteq (c \vee Q)$ **iff** $[b \Rightarrow c]$ and $P \sqsupseteq (c \vee Q)$

L7 $(v := f) \sqsupseteq (c \vee (v := g \sqcap \ldots \sqcap v := h))$ **iff** $[c \vee (f \in \{g, \ldots, h\})]$

Exercises 5.3.1

Prove the following laws if all operands are in the new normal form.

(1) $\mathbf{true} \vee (\bigsqcap A) = \mathbf{true} \vee (\bigsqcap B)$

(2) $P; (\mathbf{true} \vee (\sqcap A)) = \mathbf{true}$

(3) $(P \lhd b \rhd Q); R \;=\; (P; R) \lhd b \rhd (Q; R)$

(4) $(P \lhd b \rhd (P \lhd c \rhd Q)) \;=\; (P \lhd b \vee c \rhd Q)$

(5) $(P; \mathit{II}) \;=\; P \;=\; (\mathit{II}; P)$ □

5.4 Recursion

The introduction of recursion into the language permits construction of a program whose degree of non-determinism cannot be expressed as a finite disjunction, because it depends on the initial state. For example, let n be a non-negative integer variable in

$$\textbf{while } n \text{ is odd } \textbf{do } (n := n \ominus 1 \sqcap n := n \ominus 2)$$

where $n \ominus k$ abbreviates $0 \lhd k \geq n \rhd n - k$. Informally, the effect of this can be expressed as a disjunction of assignments

$$\begin{aligned} n &:= (n \lhd n \text{ is even} \rhd n - 1) \\ \sqcap\, n &:= (n \lhd n \text{ is even} \rhd n - 3) \\ &\vdots \\ \sqcap\, n &:= (n \lhd n \text{ is even} \rhd 0) \end{aligned}$$

But there is no *finite* set of assignments whose disjunction can replace the informal ellipses (⋮) shown above, because the length of the disjunction depends on the initial value of n.

The solution is to represent the behaviour of the program as an *infinite* sequence of expressions

$$S \;=\; \{S_i \mid i \in \mathcal{N}\}$$

Each S_i is a finite normal form, as defined in the previous section; it correctly describes all the possible behaviours of the program, but maybe some impossible ones as well. So we arrange that each S_{i+1} is potentially stronger and therefore a more accurate description than its predecessor S_i

$$(S_{i+1} \sqsupseteq S_i), \qquad \text{for all } i \in \mathcal{N}$$

This is called the descending chain condition. It allows the later members of the sequence to exclude more and more of the impossible behaviours, and in the limit, every impossible behaviour is excluded by some S_i, provided that i is large enough. Thus the exact behaviour of the program is captured by the *least upper bound* of the whole sequence, written $(\bigsqcup_i S_i)$, or more briefly $(\bigsqcup S)$. It is this that will be taken as the normal form for programs that contain recursion.

For the example shown above, we define the infinite sequence S as follows

$$F(X) =_{df} ((n := n \ominus 1 \sqcap n := n \ominus 2); X) \lhd n \text{ is odd} \rhd I\!I$$

$$\begin{aligned} S_1 &=_{df} F(\mathbf{true}) && \{5.3\mathbf{L3} \text{ and } 5.3\mathbf{L5}\} \\ &= \mathbf{true} \lhd n \text{ is odd} \rhd I\!I && \{5.3\mathbf{L2}\} \\ &= (n \geq 1 \wedge n \text{ is odd}) \vee (n := n \lhd n \text{ is even} \rhd n \ominus 1) \end{aligned}$$

$$\begin{aligned} S_2 &=_{df} F(S_1) && \{5.3\mathbf{L3}\} \\ &= ((n \geq 3 \wedge n \text{ is odd}) \vee \\ &\quad (n := n \ominus 1 \sqcap n := n \ominus 3)) \lhd n \text{ is odd} \rhd I\!I && \{5.3\mathbf{L2}\} \\ &= (n \geq 3 \wedge n \text{ is odd}) \vee \\ &\quad ((n := n \lhd n \text{ is even} \rhd n \ominus 1) \sqcap (n := n \lhd n \text{ is even} \rhd n \ominus 3)) \end{aligned}$$

$$\begin{aligned} S_{i+1} &=_{df} F(S_i) \\ &= (n \geq 2i+1 \wedge n \text{ is odd}) \vee \\ &\quad \textstyle\bigsqcap_{0 \leq k \leq i} (n := n \lhd n \text{ is even} \rhd n \ominus (2k+1)) \end{aligned}$$

Each S_i is expressible in finite normal form. It describes exactly the behaviour of the program when n is initially less than $2i$, so that the number of iterations is bounded by i. The least upper bound $\bigsqcup_i S_i$ describes the whole behaviour of the program independent of the initial value of n. This is the insight for Theorem 5.4.1 below, which is used to reduce recursions to normal form.

But first we need to provide the distribution laws that eliminate the other operators of the language. They depend critically on the descending chain condition for S, because that permits distribution of composition through least upper bound.

L1 $(\bigsqcup S) \sqcap P = \bigsqcup_i (S_i \sqcap P)$

L2 $(\bigsqcup S) \lhd b \rhd P = \bigsqcup_i (S_i \lhd b \rhd P)$

L3 $P \lhd b \rhd (\bigsqcup S) = \bigsqcup_i (P \lhd b \rhd S_i)$

L4 $(\bigsqcup S); P = \bigsqcup_i (S_i; P)$

L5 $P; (\bigsqcup S) = \bigsqcup_i (P; S_i)$, provided P is a finite normal form

Operators that distribute through least upper bounds of descending chains are called *continuous*. Every combination of continuous operators is also continuous in each of its arguments separately. In fact it is even continuous in all its arguments together. This joint continuity is assured by the descending chain condition

$$S_{i+1} \sqsupseteq S_i, \qquad \text{for all } i \in \mathcal{N}$$

A descending chain of descending chains can be reduced to a single descending chain by diagonalisation

$$\bigsqcup_k (\bigsqcup_l S_{k,l}) \;=\; \bigsqcup_i S_{i,i}$$

provided that

$$(S_{k,i+1} \sqsupseteq S_{k,i}) \text{ and } (S_{i+1,l} \sqsupseteq S_{i,l}), \qquad \text{for all } i, k \text{ and } l$$

because then $S_{k,l} \sqsubseteq S_{i,i}$, where $i = max(k, l)$. This ensures that a function F, continuous in each of its arguments separately, is also continuous in its arguments as they tend to their limits simultaneously

$$F(\bigsqcup S, \bigsqcup T) \;=\; \bigsqcup_i F(S_i, T_i)$$

In turn, this gives the required elimination laws to compute normal forms for the three operators of the language.

L6 $(\bigsqcup S) \sqcap (\bigsqcup T) \;=\; \bigsqcup_i (S_i \sqcap T_i)$

L7 $(\bigsqcup S) \lhd b \rhd (\bigsqcup T) \;=\; \bigsqcup_i (S_i \lhd b \rhd T_i)$

L8 $(\bigsqcup S); (\bigsqcup T) \;=\; \bigsqcup_i (S_i; T_i)$

The occurrence of the operators $\sqcap$, $\lhd b \rhd$ and ; on the right hand side of these laws can be eliminated by the laws of Section 5.3, since each S_i and T_i is finite.

The continuity laws ensure that descending chains constitute a valid normal form for all the combinators of the language, and the stage is set for treatment of recursion. Consider first an innermost recursive program (containing no other recursions)

$$\mu X \bullet F(X)$$

where $F(X)$ contains X as its only free recursive identifier. The recursive identifier X is certainly not in normal form, and this makes it impossible to express $F(X)$ in normal form. However, all the other components of $F(X)$ are expressible in finite normal form, and all its combinators permit reduction to finite normal form. So, if X were replaced by a normal form (say **true**), F(**true**) can be reduced to finite normal form, and so can $F(F(\mathbf{true}))$, $F(F(F(\mathbf{true})))$, ... Furthermore, because F is monotonic, this constitutes a descending chain of normal forms. Since F is continuous, by Kleene's famous recursion theorem, the limit of this chain is the least fixed point of F

Theorem 5.4.1 (Kleene)

If F is continuous then

$$\mu X \bullet F(X) \;=\; \bigsqcup_n F^n(\mathbf{true})$$

where $F^0(X) =_{df} \mathbf{true}$, and $F^{n+1}(X) =_{df} F(F^n(X))$. □

This reduction can be applied first to replace all the innermost recursions in the program by limits of descending chains. The treatment of the remaining innermost recursions needs a continuity law for recursion, which enables each recursion to be eliminated as soon as it contains no other recursion.

Theorem 5.4.2 (Continuity of μ)

$$\mu X \bullet \bigsqcup_i S_i(X) \;=\; \bigsqcup_i \mu X \bullet S_i(X)$$

provided that $S_i(X)$ contains X as its only free recursive identifier for all i, and that all S_i are continuous and they form a descending chain for all finite normal forms X, that is

$$(S_{i+1}(X) \sqsupseteq S_i(X)), \qquad \text{for all } i \in \mathcal{N}$$

Proof Let $F(X) =_{df} \bigsqcup_i S_i(X)$. We are going to show by induction that for all $n \geq 0$

$$F^n(\mathbf{true}) \;=\; \bigsqcup_i S_i^n(\mathbf{true})$$

For $n = 0$

$$F^0(\mathbf{true}) \;=\; \mathbf{true} \;=\; \bigsqcup_i \mathbf{true} \;=\; \bigsqcup_i S_i^0(\mathbf{true})$$

By induction for any $n > 0$

$$\begin{aligned}
& F^n(\mathbf{true}) && \{F^{k+1}(X) = F(F^k(X))\} \\
=\; & \bigsqcup_i S_i(F^{n-1}(\mathbf{true})) && \{\text{induction hypothesis}\} \\
=\; & \bigsqcup_i S_i(\bigsqcup_j S_j^{n-1}(\mathbf{true})) && \{S_i \text{ is continuous}\} \\
=\; & \bigsqcup_i \bigsqcup_j S_i(S_j^{n-1}(\mathbf{true})) && \{\text{diagonalisation}\} \\
=\; & \bigsqcup_i S_i(S_i^{n-1}(\mathbf{true})) && \{\text{def of } S_i^n\} \\
=\; & \bigsqcup_i S_i^n(\mathbf{true})
\end{aligned}$$

The function F is also continuous because all S_i are continuous, and they form a descending chain for all X. By Theorem 5.4.1 we have

$$\mu X \bullet \bigsqcup_i S_i(X) \;=\; \bigsqcup_n F^n(\mathbf{true}) \;=\; \bigsqcup_n \bigsqcup_i S_i^n(\mathbf{true}) \;=\; \bigsqcup_i \mu X \bullet S_i(X) \qquad \square$$

Let us now return to the task of eliminating recursions from the program.

Suppose all the innermost recursions have been replaced by limits of descending chains. The remaining innermost recursions now have the form

$$\mu Y \bullet H(\sqcup_m \ F^m(\mathbf{true}), \ \sqcup_m \ G^m(\mathbf{true}), \ldots, Y)$$

By continuity of H, this transforms to

$$\mu Y \bullet \sqcup_m \ H_m(Y)$$

where $H_m(Y) =_{df} H(F^m(\mathbf{true}), \ G^m(\mathbf{true}), \ldots, Y)$, which is (for fixed Y) a descending chain in m. By Theorem 5.4.2, this equals

$$\sqcup_m \ \mu Y \bullet H_m(Y)$$

and by Kleene's theorem

$$\sqcup_m \sqcup_n \ (H_m^n(\mathbf{true}))$$

Because this is descending in both n and m, we get at last

$$\sqcup_n \ H_n^n(\mathbf{true})$$

Thus the next innermost recursions are converted to normal form; by repeating the process, the whole program can be converted to normal form

$$\sqcup_n \ S_n$$

Another way of describing the same conversion is that S_n is the result of replacing *every* recursion $\mu X \bullet F(X)$ in the program by the n^{th} element of its approximating series, that is $F^n(\mathbf{true})$.

The finite normal forms play a role similar to that of rational numbers among the reals. Firstly, there is only a countable number of them. A second similarity is that every real is the limit of a descending chain of rationals. Finally, the rationals are dense in reals, in the sense that any two distinct real numbers can be shown to be so by a rational number which separates them. The application of these insights to computer programs is the contribution of Scott's theory of continuous domains [165, 167].

The definition of a testing criterion for infinite normal forms is postponed to Section 5.6.

Exercises 5.4.3

Prove from the laws of this chapter that

(1) $\mathbf{true}; P \ = \ \mathbf{true}$

(2) If $P \sqsupseteq F(P)$ then $P \sqsupseteq \mu F$. □

5.5 Iteration

Iteration has already been defined in Section 2.6 as a special case of recursion (tail recursion)

$$b * Q \;=_{df}\; \mu X \bullet (Q; X) \lhd b \rhd I\!I$$

On a machine with limited resources it is implemented more economically than general recursion. Furthermore, reasoning about iteration can be simpler than the general case. This section develops the simpler theory of iteration through a series of lemmas that will be useful in later chapters. The proofs will use algebraic reasoning on the basis of the normal form theorem. The theorems are in fact valid for all designs, not just those that can be expressed in normal form.

If condition b is true initially, the iteration $(b * Q)$ executes Q first, and then proceeds like $(b * Q)$. Otherwise it terminates immediately.

L1 $b_\perp; (b * Q) \;=\; b_\perp; Q; (b * Q)$ (loop unfold)

L2 $(\neg b)_\perp; (b * Q) \;=\; (\neg b)_\perp$ (loop elim)

If the execution of Q always establishes the condition b, then $(b * Q)$ behaves like the assertion $(\neg b)_\perp$.

L3 $(b * Q) \;=\; (\neg b)_\perp,$ provided that $(Q; b_\perp) \;=\; Q$ (loop abort)

The next law enables us to transform a tail recursion to an iteration.

L4 $\mu X \bullet (P; X) \lhd b \rhd Q \;=\; (b * P); Q$ (tail recursion)

Proof Define

$$F(X) \;=_{df}\; (P; X) \lhd b \rhd Q$$
$$G(X) \;=_{df}\; (P; X) \lhd b \rhd I\!I$$

We are going to show by induction that

$$F^n(\mathbf{true}) \;=\; G^n(\mathbf{true}); Q$$

When $n = 0$ we have

$$\begin{array}{ll} & F^0(\mathbf{true}) & \{\text{def of } F^0\} \\ = & \mathbf{true} & \{\text{Exercise 5.4.3(1)}\} \\ = & (\mathbf{true}; Q) & \{\text{def of } G^0\} \\ = & G^0(\mathbf{true}); Q & \end{array}$$

By induction for any $n \geq 0$ one has

$$\begin{aligned}
& F^{n+1}(\mathbf{true}) && \{\text{def of } F^{n+1}\} \\
= \; & (P; F^n(\mathbf{true})) \lhd b \rhd Q && \{\text{induction hypothesis}\} \\
= \; & (P; G^n(\mathbf{true}); Q) \lhd b \rhd Q && \{\text{Exercises 5.3.1(3) and 5.3.1(5)}\} \\
= \; & ((P; G^n(\mathbf{true})) \lhd b \rhd \mathit{II}); Q && \{\text{def of } G^{n+1}\} \\
= \; & G^{n+1}(\mathbf{true}); Q
\end{aligned}$$

The conclusion then follows from Kleene's theorem and the left continuity of sequential composition. □

Corollary $b * Q \;=\; (b * Q); (\neg b)_\perp$ (loop termination)

The following law allows the combination of loops with the same body.

L5 $(b * Q); (b \vee c) * Q \;=\; (b \vee c) * Q$ (merge of loop)

Proof Define $S(X) =_{df} (Q; X) \lhd b \rhd (b \vee c) * Q$. From **L4** it follows that

$$LHS = \mu S$$

so we only need to show $(b \vee c) * Q \;=\; \mu S$.

$$\begin{aligned}
& (b \vee c) * Q = ((b \vee c) * Q) \lhd b \rhd ((b \vee c) * Q) && \{\mathbf{L1}\} \\
\Rightarrow \; & (b \vee c) * Q = (Q; (b \vee c) * Q) \lhd b \rhd ((b \vee c) * Q) && \{\text{weakest fixed point}\} \\
\Rightarrow \; & (b \vee c) * Q \sqsupseteq \mu X \bullet ((Q; X) \lhd b \rhd (b \vee c) * Q) = \mu S && \\
& \text{fixed point theorem} && \\
\Rightarrow \; & \mu S = (Q; \mu S) \lhd b \rhd (b \vee c) * Q && \{\text{fixed point}\} \\
\Rightarrow \; & \mu S = (Q; \mu S) \lhd b \rhd ((Q; (b \vee c) * Q) \lhd b \vee c \rhd \mathit{II} && \{(b \vee c) * Q \sqsupseteq \mu S\} \\
\Rightarrow \; & \mu S \sqsupseteq (Q; \mu S) \lhd b \rhd ((Q; \mu S) \lhd b \vee c \rhd \mathit{II}) && \{\text{Exercise 5.3.1(4)}\} \\
\Rightarrow \; & \mu S \sqsupseteq (Q; \mu S) \lhd b \vee c \rhd \mathit{II} && \{\text{weakest fixed point}\} \\
\Rightarrow \; & \mu S \sqsupseteq \mu X \bullet ((Q; X) \lhd b \vee c \rhd \mathit{II}) = (b \vee c) * Q && \square
\end{aligned}$$

Exercises 5.5.1

(1) $b * P \;=\; b * (P; b * P)$

(2) If $c_\perp; P \;=\; c_\perp; P; c_\perp$ then $c_\perp; (b * P) \;=\; c_\perp; (b * P); c_\perp$. □

5.6 Computability

The algebraic laws given in Sections 5.1, 5.2 and 5.3 permit every finite program (one that does not use recursion) to be reduced to finite normal form. The reduction rules are nothing but simple algebraic transformations, of the kind that can be readily mechanised on a computer. The infinite normal form $(\bigsqcup_i S_i)$ of Section 5.4 can never be computed in its entirety; however, for each n, the finite normal form S_n can be readily computed, for example by replacing each internal recursion $(\mu X \bullet F(X))$ by $(F^n(\mathbf{true}))$.

This suggests a highly impractical method of executing a program, starting in a known initial state s, in which Boolean conditions can be evaluated to *true* or *false*. The machine calculates the series S_n of finite normal forms from the program. Each of these is a disjunction $(b_n \vee P_n)$. If $(s; b_n)$ evaluates to *true*, the machine continues to calculate the next S_{n+1}. If *all* the $(s; b_n)$ are *true*, this machine never terminates, but that is the right answer, because in this case the original program, when started in the given initial state s, contains an infinite recursion or loop. But as soon as a false $(s; b_n)$ is encountered, the corresponding P_n is executed, by selecting and executing an arbitrary one of its constituent assignments. We want to prove that the resulting state will be related to the initial state as described by *all* the S_i, when these are interpreted as predicates in accordance with the definitions of Chapter 2.

The validity of this method of execution depends on an additional property of the normal form, that $\{b_n \vee P_n\}$ is a descending chain in a stronger ordering, where comparability requires two terminating programs to be exactly the same in their range of non-determinism

$$(b_n \vee P_n) \equiv (b_n \vee P_{n+k}), \qquad \text{for all } n, k$$

that is once n is high enough for b_n to be false, all the assignments P_m remain the same as P_n, for all m greater than n. Let us therefore define a new ordering relation.

Definition 5.6.1 (Strong ordering)

Let $P =_{df} (v := e_1 \sqcap \ldots \sqcap v := e_m)$ and $Q =_{df} (v := f_1 \sqcap \ldots \sqcap v := f_n)$.
Define

$$(b \vee P) \leq (c \vee Q) =_{df} [c \Rightarrow b] \wedge [\neg b \Rightarrow (\{e_1, \ldots, e_m\} = \{f_1, \ldots, f_n\})] \quad \square$$

The new ordering is stronger than the familiar implication ordering.

Theorem 5.6.2

If $(b \vee P) \leq (c \vee Q)$ then $(b \vee P) \sqsubseteq (c \vee Q)$. $\square$

Strong ordering is clearly a preorder, with weakest element **true**. What is more, all the combinators of the programming language are monotonic with respect to $\leq$. If $F(X)$ is a program, it follows that $\{F^n(\mathbf{true}) \mid n \in \mathcal{N}\}$ is a descending chain in this new ordering. This shows that all innermost recursions enjoy the strong descending chain condition. Furthermore, because of continuity, any program combinator preserves this condition. For nested recursions, the proof of the normal form reduction theorem uses the same construction as given at the end of Section 5.4. All the chains involved are descending in the strong ordering as well.

We can use the following laws to compare least upper bounds of strong descending chains.

L1 $(\bigsqcup S) \sqsubseteq (\bigsqcup T)$ **iff** $\forall i : i \in \mathcal{N} \bullet S_i \sqsubseteq (\bigsqcup T)$

L2 Let $T = \{(c_n \vee Q_n) \mid n \in \mathcal{N}\}$ be a descending chain in the strong ordering, and $Q_n = (v := f_1^n \sqcap \ldots \sqcap v := f_{k_n}^n)$ for all $n \in \mathcal{N}$. Then

$(b \vee (v := e_1 \sqcap \ldots \sqcap v := e_m)) \sqsubseteq (\bigsqcup T)$ **iff**

$[(\bigwedge_n c_n) \Rightarrow b]$ and $\forall n : n \in \mathcal{N} \bullet [c_n \vee b \vee (\{f_1^n, \ldots, f_{k_n}^n\} \subseteq \{e_1, \ldots, e_m\})]$

Exercise 5.6.3

Prove that all programming operators are monotonic with respect to $\leq$. □

5.7* Completeness

A reduction to normal form gives a method of testing the truth of any proposed implication between any pair of programs: reduce both of them to normal form, and test whether the inequation satisfies the simpler conditions laid down in 5.6**L1** and 5.6**L2** for implication of normal forms. If so, it holds also between the original programs. This is because the reduction laws only substitute equals for equals, and each of the tests for implication between normal forms has been proved as a theorem. The procedure has been described in detail for the assignment normal form in Section 5.1, but the same idea works just as well for the more general forms introduced in later sections.

For the algebra of programs, the converse conclusion can also be drawn: if the test for implication fails for the normal forms, then the implication does *not* hold between the original programs. The reason is that the tests give both necessary and sufficient conditions for the validity of implication between normal forms. For this reason, the algebraic laws are said to be *complete*. Of course, since the normal form is infinite, there cannot be any general decision procedure; comparison of recursive programs will always require proof.

Theorem 5.7.1 (Completeness)

For all programs P and Q

$$[P \Rightarrow Q] \quad \textbf{iff} \quad P \sqsupseteq Q \text{ has an algebraic proof} \qquad \square$$

Completeness is a significant achievement for a theory of programming. Each of the laws requires a non-trivial proof, involving full expansion of the definitions of all the operators in the formulae, followed by reasoning in the rather weak framework of the predicate calculus. But after a complete set of laws have been proved in this more laborious way, proof of any additional laws can be achieved by purely algebraic reasoning; it will never be necessary again to expand the definitions. For example, we prove the right zero law

$$P ; \mathbf{true} = \mathbf{true}$$

Since P is a program, it can be reduced to the normal form $\bigsqcup S$.

$$\begin{array}{lll} & P; \mathbf{true} & \\ = & (\bigsqcup S); \mathbf{true} & \{\text{for some } P_i,\ b_i\} \\ = & \bigsqcup_i (b_i \vee P_i); \mathbf{true} & \{5.4\mathbf{L4} \text{ and } 5.3\mathbf{L3}\} \\ = & \bigsqcup_i ((b_i \vee (P_i; \mathbf{true})) \vee P_i) & \{5.3\mathbf{L4} \text{ and } \mathbf{L5}\} \\ = & \bigsqcup_i (\mathbf{true} \vee P_i) & \{\text{Exercise } 5.3.1(1)\} \\ = & \mathbf{true} & \end{array}$$

Apart from the practical advantages, completeness of the laws has an important theoretical consequence in characterising the nature of the programming language. For each semantically distinct program there is a normal form with the same meaning, and this can be calculated by application of the laws. It is therefore possible to regard the normal form itself as a definition of the meaning of the program, and to regard the algebraic laws as a definition of the meaning of the programming language, quite independent of the interpretation of programs as predicates describing observations. This is the philosophy of "initial algebra" semantics for abstract data types [68].

There are many advantages in this purely algebraic approach [16, 117]. It is often quite easy to decide what laws (like the zero laws) are needed or wanted for a programming language, and then it is much easier just to postulate them than to prove them. Furthermore, algebra is the most reusable branch of mathematics. Most algebraic laws are valid for many different programming languages, just as most of conventional schoolroom algebra holds for many different number systems. Even the differences between the number systems are most clearly described and understood by examining the relatively simple differences in their algebraic presentations. That is certainly clearer than the widely differing definitions which they

are given in the foundations of mathematics. In the study of the foundations of mathematics, no useful comparison can be made between the classes of sets which define the integers, and the Dedekind cuts which define the reals; only algebra provides the right level of abstraction to compare them effectively and to justify the use of the same symbols to denote the arithmetic operators.

In practical engineering, algebraic laws are widely used in symbolic calculation of parameters and design details, which are derived as consequences of more general structural decisions made by engineering judgement. The algebraic laws neatly encapsulate the engineers' experience and intuition, making them easy to learn, remember and apply. Computer support for algebraic reasoning is far better developed than that for general proof search. In the division of labour between engineers and mathematicians, it is the role of the engineer to calculate and the mathematician to prove, and the proof should establish not only the correctness but also the completeness of the algebra used in calculation.

But there is an alternative approach to the theory of programming, which recognises the greater importance of the laws in the practical application of the theory; it therefore presents the laws by themselves as postulates or axioms; they have to be accepted as self-evident, or at least as an indirect definition of the meaning of the notations involved; they are then used as the basis for all general and particular reasoning about programs. This axiomatic approach is taken in the study of all branches of modern algebra, and has been adopted in the presentation of many versions of programming language semantics [18, 85].

If the algebra is taken as the official starting point for the investigation, the direction of the development of the theory has to be reversed. Instead of deriving a complete set of algebraic laws from the denotational semantics, it is necessary to derive a denotational semantics from the laws. That is the task of the last section of this chapter. It is a matter of personal choice whether the investigation of algebra precedes the search for possible meanings, or the other way round (as in this book). The experienced mathematician probably explores both approaches at the same time. When the task is complete, the practising engineer or programmer has a secure intellectual platform for understanding complex phenomena, and a full set of calculation methods for the reliable design of useful products. That is the ultimate, if not the only, goal of our investigations.

5.8* From algebraic to denotational semantics

In this section, we need to start with a more traditional presentation of a denotational semantics, which explicitly associates with each program the set of observations to which it can give rise, as described in Section 1.2. The set is a relation between the initial and final states of any execution of the program. An

observation of the *final* state of a program is coded in the manner described in Section 1.1, as an equation

$$state := k \qquad \text{(defined as the predicate } state' = k\text{)}$$

where k is a list of constants, and *state* consists of variable *ok* and program variables v. Similarly, if j is a list of constants,

$$(state := j); P(state, state')$$

describes all possible observations of the final state of an execution of P that starts in *initial* state j. The implication

$$[(state := k) \Rightarrow ((state := j); P)]$$

means the state k is among the possible final states of a program execution that starts in j. In fact, using the definition of composition, it is exactly equivalent to the result of substitution of constants for all free variables

$$P(j, k)$$

Because the only free variables of our predicates are those that describe initial and final states, it is equally trivial to see that one predicate P implies another Q just if its set of initial and final states is a subset of the initial and final states of the other

$[P \Rightarrow Q]$ **iff**

$[t \Rightarrow (s; P)]$ implies $[t \Rightarrow (s; Q)]$, for all constant assignments s and t

This is the insight that links our treatment of programs as predicates with a more traditional denotational semantics. The latter is given as a function $\mathcal{R}$ mapping every program onto a relation between its initial and final states

$$\mathcal{R}(P) = \{(j, k) \mid [(state := k) \Rightarrow (state := j; P)]\}$$

From this definition, one can derive a collection of equations relevant to each of the constructions of the programming language.

$$\begin{aligned}
\mathcal{R}(v := e(v)) &= \{((ok, v), (ok', v')) \mid ok \Rightarrow (ok' \wedge v' = e(v))\} \\
\mathcal{R}(P; Q) &= \{(j, k) \mid \exists u \bullet (j, u) \in \mathcal{R}(P) \wedge (u, k) \in \mathcal{R}(Q)\} \\
\mathcal{R}(P \sqcap Q) &= \{(j, k) \mid (j, k) \in \mathcal{R}(P) \vee (j, k) \in \mathcal{R}(Q)\} \\
\mathcal{R}(P \lhd b \rhd Q) &= \{(j, k) \mid ((state := j); b) = true \wedge (j, k) \in \mathcal{R}(P) \vee \\
&\qquad ((state := j); b) = false \wedge (j, k) \in \mathcal{R}(Q)\} \\
\mathcal{R}(\mu X \bullet F(X)) &= \{(j, k) \mid \forall n : \mathcal{N} \bullet (j, k) \in \mathcal{R}(F^n(\mathbf{true}))\}
\end{aligned}$$

A denotational semantics in the traditional sense takes those equations as a *definition* of the function $\mathcal{R}$, which maps each program text to its mathematical meaning as a relation. The equational definition of $\mathcal{R}$ appeals to the principle of structural induction over the texts of programs expressed in the limited notations of the programming language: there is only one function that satisfies all the equations. This principle gives an important proof method: any independently defined function $\mathcal{S}$ which is proved to satisfy these equations is necessarily equal to $\mathcal{R}$.

This insight gives a sufficient basis for deriving a denotational semantics from the algebraic. All that is needed is to use the *algebra* to associate with each program P an appropriate set $\mathcal{S}(P)$ of pairs of constant assignments. The definition uses provability in the algebra to define the appropriate set.

Definition 5.8.1

$$\mathcal{S}(P) =_{df} \neg d(v) \vdash R(v, v')$$

where $d(m) =_{df} ((v := m; P) = \mathbf{true})$ is algebraically provable

and $R(m, n) =_{df} ((v := m; P) \sqsubseteq (v := n))$ is algebraically provable. □

Theorem 5.8.2

$\mathcal{S}$ satisfies all the equations used to define $\mathcal{R}$.

Proof Based on structural induction and the following facts that for any program P and any constant assignment $v := m$

either $(v := m; P) = \mathbf{true}$ is algebraically provable

or there exists a finite set of constant assignments $\{v := n \mid n \in T\}$ such that the following is algebraically provable

$$(v := m; P) = \bigsqcap \{v := n \mid n \in T\}$$ □

Exercise 5.8.3

Fill in the missing clauses $\mathit{\Pi}$ and **true** in the definition of $\mathcal{R}$. □